ESQL : an extended SQL with object and deductive capabilities

Georges Gardarin, Patrick Valduriez

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1185

Programme 4
Bases de Données

ESQL : AN EXTENDED SQL WITH OBJECT AND DEDUCTIVE CAPABILITIES

Georges GARDARIN
Patrick VALDURIEZ

Mars 1990

RR_1185

ESQL : Une Extension de SQL avec des Possibilités Objets et Déductives[1]

Georges GARDARIN & Patrick VALDURIEZ

INRIA & MASI

BP.105, 78153 Le Chesnay-Cédex

Résumé

ESQL est un langage de base de données compatible avec SQL qui intègre uniformément les concepts des bases de données relationnelles, orientées objets et déductives. ESQL devrait supporter aussi bien les applications de gestion classique que les applications plus complexes telles que de grands systèmes experts. Les éléments essentiels de ESQL sont donc : un système de types riche et extensible grâce à des types abstraits de données (TADs) implémentés dans divers langages ; les objets complexes avec partage référentiel en combinant les TADs génériques et l'identité d'objet ; la manipulation de relations contenant des objets simples ou complexes avec une syntaxe et une sémantique compatible avec SQL ; et une possibilité de déduction de type DATALOG offerte comme une extension du mécanisme de vues. La sémantique fonctionnelle de ESQL uniformise la manipulation des données et devrait faciliter le développement du compilateur ESQL.

1 Ce travail est financé par le projet ESPRIT EDS.

ESQL : An Extended SQL with Object and Deductive Capabilities[1]

Georges GARDARIN & Patrick VALDURIEZ

INRIA & MASI

BP 105, 78153 LE CHESNAY

ABSTRACT

ESQL is an SQL upward-compatible database language that integrates in a uniform and clean way the essential concepts of relational, object-oriented and deductive databases. ESQL is intended for traditional data processing applications as well as more complex applications such as large expert systems. Therefore, ESQL's salient features are: a rich and extendible type system based on abstract data types (ADTs) implemented in various programming languages; complex objects with object sharing by combining generic ADTs and object identity; the capability of querying and updating relations containing simple or complex objects using SQL-compatible syntax and semantics; and a DATALOG-like deductive capability provided as an extension of the SQL view mechanism. ESQL's functional semantics enables uniform manipulation of ESQL data and should facilitate the implementation of the ESQL compiler.

[1] This work is sponsored by ESPRIT project EDS

1. INTRODUCTION

The relational database approach has gained wide acceptance in traditional data processing (i.e., business) mainly because it increases user productivity and enables automatic optimization of database accesses and updates. This approach is highly promoted by the existence of the standard relational query language SQL [ISO86] which provides a uniform interface to database administrators, application programmers and end-users for data definition and data manipulation. As a result, SQL is becoming the common language for exchanging data in centralized, decentralized and heterogeneous environments.

Current relational database systems have been designed for traditional data processing applications. Therefore, they do not support well emerging database application domains such as computer aided design (CAD), office information systems (OIS) and knowledge-based systems (KBS), in particular expert systems. These applications express new requirements such as user-defined data types (incorporating both data structures and their associated operations), complex objects (identified values of rich type or complex structure) and rule-based knowledge management. Research aimed at supporting new application requirements in a more integrated way led to two new approaches : object-oriented databases and deductive databases [Gardarin89].

Object-oriented databases aim primarily at supporting user-defined data types and complex objects. Their salient features are abstract data types (ADTs) [Guttag77], object identity, type inheritance and persistence independence [Bancilhon88]. ADTs enable the encapsulation of data and their associated operations (called methods) while hiding implementation details. Object identity [Khoshafian86] provides referential object sharing [Khoshafian87] to model complex (graph-structured) objects without replication. Type inheritance enables objects of different types to share the operations pertinent to their common supertypes thereby encouraging reuse of code and reducing the size of application programs. The object-oriented database approach combines object-oriented programming, as exemplified by C++, and database technology. Therefore, database applications can be entirely written in a single

database programming language (e.g., as opposed to SQL embedded in a programming language).

Deductive databases focus on integrating within the database the knowledge typically embedded in application programs. With better sharing, control and management of knowledge, typically represented by assertions and deductive rules, deductive databases can reduce significantly the size and complexity of application programs. The deductive database approach combines logic programming, as exemplified by PROLOG, and database technology. Therefore, it provides a highly expressive database language which allows powerful queries such as recursive queries.

These approaches bring definite advantages over relational databases. However, they generally come with what is perceived by traditional database users as a new language (although it is more a significant extension of an object-oriented or logic programming language). This hampers the acceptance of these technologies by data processing users who could also benefit from their advanced features. Furthermore, albeit it is Turing-complete, a new language does not necessarily solve the infamous "impedance mismatch" problem [Copeland84] for the simple reason that there will always be users who want to access the database from their favorite programming language. In other words, the impedance mismatch is not a model or language problem, but must be dealt with by the database system.

A more conservative approach capitalizes on relational database technology and extends it in a way that the advantages of object-oriented databases and deductive databases can be gained. The rationale behind this approach is that the relational model provides a stable platform with simple concepts open for such extensions. The concept of nested relation [Ozsoyoglu87] is useful to model hierarchical structures and manipulate them with a SQL-like language [Schek88]. The concept of domain, not constrained in its original definition, can be defined as ADT [Stonebraker83, Gardarin89b], query or procedure [Stonebraker86], or object [Carey88, Danforth88]. In addition, a relational query language can be given sound functional semantics, e.g., OSQL [Beech88] and FSQL [Valduriez89a], and thus support powerful user-defined functions. Each of these extensions provide some generality over the relational model. However, they have been proposed independently and in various contexts, thereby making their integration difficult.

The contribution of this paper is to integrate in a uniform and simple way the key concepts of object-oriented databases and deductive databases within the SQL framework. The result is ESQL (for Extended SQL), an SQL upward-compatible query language supporting rich types, complex objects and derived relations. For instance, these capabilities are essential for large expert system applications that typically require inferencing (deduction) on complex objects. Rich type support is provided by a generic ADT capability allowing multiple implementation programming languages. Complex object support is provided by structures that may contain arrangement of values of any types. Finally, recursive derived relations are supported through an extension of the SQL view mechanism. Furthermore, ESQL's functional semantics make uniform and simple the manipulation of ESQL data and should ease the development of an ESQL compiler [Valduriez89b]. Finally, ESQL's relational basis favors reuse of known query optimization techniques [Valduriez89c].

The following presentation concentrates on the most powerful concepts of ESQL. Section 2 gives an overview motivating the design of ESQL. Section 3 introduces a powerful ADT model with ADT inheritance and ADT constructors. This model provides a solid basis for supporting efficiently objects of rich type and complex structure. Section 4 illustrates the data definition and data manipulation facilities of ESQL. Section 5 presents the ESQL derived relation capability. Finally, Section 6 introduces in informal terms the functional semantics of ESQL.

2. ESQL OVERVIEW

ESQL is intended for traditional data processing applications written in standard SQL as well as non-traditional data intensive applications. To promote the access of SQL users to ESQL's new functions, the language extension is provided with minimal impact to the SQL syntax. The main advantages provided by ESQL over SQL are strong support for abstract data types specified in different programming languages, complex objects with object sharing, and a deductive capability to infer new data from stored data.

The support of ADTs provides a rich typing capability. It makes the fixed set of system-defined types extendible by the users to accommodate their application specific requirements. An ADT is a new type together with

methods applicable to data of that type. The value of any ADT (e.g., map) can be stored in the database system and manipulated using the associated methods (e.g., map intersection). ESQL supports this ADT capability by extending the notion of domain traditionally supported by relational database systems. The goal is to allow ADTs to be implemented in various languages such as C, C++, LISP and PROLOG. For each ADT implementation language supported, the database system must provide routines to convert between the language data structures (e.g., list in LISP) and a supporting database system type (e.g., string).

ESQL's support for complex objects includes object identity, which enables objects to be referentially shared, and permits the construction of complex structures (e.g., hierarchy, graph). Complex objects such as office automation objects or CAD design objects can be modelled and manipulated in a natural way, while giving an ESQL compiler opportunities to optimize the access to such objects. The relational data model only supports values, imposing the use of key values to identify objects. ESQL supports both values and objects. A value is an instance of an ADT while an object has a unique identifier with a value bound to it. Data not declared as objects are values by default as in the relational model, which means that they have no system identifier. Therefore, ESQL data are divided between objects and values, and only objects may be referentially shared using object identity. The relational data model supports flat values using the tuple and set constructors at one level. To support complex values, ESQL generalizes the relational model with a library of generic ADTs which may be combined at multiple levels. Generic ADTs are higher-order constructors that take as arguments values or objects of any type. The primary generic ADTs are tuple, set, bag, list and array. Others useful constructors are ordered sequence and graph. By combining objects and generic ADTs, arbitrarily complex objects can be supported.

A deductive capability enables one to abstract in a rule base the common knowledge traditionally embedded with redundancy in application programs. The rule base provides centralized control of knowledge, and is primarily useful to infer new facts from the facts stored in the database. ESQL provides this deductive capability as an extension of the SQL view mechanism. This gives the ESQL user the power of the DATALOG logic-based language using statements already available in SQL.

The ESQL data definition language (DDL) augments the SQL DDL in three major ways. First, it includes a type language to create and manage ADTs with related methods, and generic ADTs. Second, it extends the table creation statement to deal with nested relations and objects. Third, it adds a statement to create derived relations defined by general rules, including recursive rules.

The ESQL data manipulation language (DML) generalizes the SQL DML in several ways. The most significant advantages are the possibility of ADT operations in SQL statements, the manipulation of shared objects by extending the dot notation, the manipulation of nested objects through nested statements and the possibility of deductive queries. Data manipulation in ESQL is more regular than SQL, much in the way of SQL2 [Melton89].

3. ABSTRACT DATA TYPE DEFINITION

3.1 Overview of the Data Model

The data model of SQL is relational in nature, and therefore based primarily on two concepts : domains of values and relations on domains. The data model of ESQL is the relational model extended with abstract data types (ADTs), which generalize the domains of values. Thus, a domain of value in a relation is defined either as an elementary data type or as an ADT. An elementary data type provides a domain of values directly supported by standard SQL2 (i.e., character strings, numbers, enumerated types, dates, times, intervals and nulls) and comes with operations applicable to values of that type, for instance, arithmetics on numbers. Similarly, an ADT defines a domain of values, for instance, text or triangle, and comes with user-defined operations, for instance, extract the title from a text or compute the surface of a triangle.

An ADT is viewed as a set of methods (or functions) that operate on values of the defined type. Thus, an ADT encapsulates the type structure within a set of operations so that the implementation details are hidden from the ADT user who only sees the interface methods. Values of a given ADT are manipulated through functions (known as methods). Public methods can be invoked within DML statements, both in the definition of query results, query

qualification and update actions. A specific command (CREATE TYPE) is added to SQL to enter the abstract data type definition toolbox and define a new ADT.

An ADT structure is built by combining elementary data types or other ADTs using generic ADTs (i.e., generalized constructors). The primary generic ADTs needed to achieve functionalities required by engineering or office applications are supplied by a system tool box to define ADTs. This includes the tuple, set, bag, list, array generic data types (sequence and graph could also be added). All user defined ADTs are built from this set of generic ADTs and a base set of simple type. The usual notion of type inheritance is supported : an ADT can be a subtype of another ADT; it then inherits all the methods of the generic ADT. Further, new attributes can be added to a tuple inherited type. In general, methods can be redefined (i.e., overloading is permitted) or added to a subtype. For instance, triangle may be a subtype of polygon. A height attribute can then be added to the triangle subtype; heights should then be updated by user programs. An alternative is to define height as a method that computes the triangle height from the sides.

3.2 Generic Abstract Data Types

An ADT specification is a representation independent functional definition of each operation of an ADT. This specification will be the only visible part of the ADT in SQL. Thus, the complete design of an ADT would proceed by first giving its specification, followed by an implementation that agrees with the specification.

ESQL includes a set of generic ADTs to specify and implement ADTs. Generic ADTs are useful for specialization. Generic ADTs are typically parametrized by one or more types. A generic ADT capability is a powerful tool for constructing new ADTs; it offers a homogeneous implementation of constructors as tuple and collections. Collections may be specialized in sets, bags, list, or vectors, which are specific generic ADTs of most object-oriented database systems.

In the following, we define the signature of specific generic ADTs. For this purpose, we introduce a few notations. The notation "[a]" indicates that a is optional. The notation "a | b" indicates that a or b must be chosen. The notation "{a}... " indicates that a is repeated one or more times. A function is

defined as F(x,...) --> y, where x,... specifies the arguments type and y the result type. ø is the empty type, which means no element.

A simple example of ADT is the tuple ADT. Denoting by a1, a2, ... values of type Type1, Type2, ..., denoting by A1, A2, ... the attribute names and assuming that Formula is a logical formula on A1, A2, ..., the tuple ADT may be specified as follows :

```
ADT TUPLE OF (<A1> : <Type1> [ { , <A2> : <Type2> } ... ] )
        FUNCTION :
                MAKETUPLE(a1 [ {, a2} ...] --> Tuple
                EQUAL(Tuple,Tuple) --> Boolean
                PROJECT(Tuple, A1, A2 ...) --> Tuple
                ASSIGN(Tuple,A1,A2,..., a1,a2,...) --> Tuple
                CHECK(Tuple, Formula) --> Boolean
                PRODUCT(Tuple,Tuple) --> Tuple
        END TUPLE ;
```

MAKETUPLE creates a tuple of the generic type with initial values. EQUAL checks the equality of two tuples. PROJECT is the projection function. ASSIGN assigns the given attribute values to the corresponding attributes of the tuple. CHECK checks whether a given tuple satisfies the parameter formula. PRODUCT performs the Cartesian product of two tuples.

In ESQL, special attention is given to the support of collections. Collection is a built-in generic ADT in the language. Collections are organized in an inheritance hierarchy whose root is Collection and subtypes are bag, set, list and vector (see Figure 1). Certain functions, such as COUNT, ISEMPTY, EQUAL, INSERT, REMOVE are general and act on any type of collections: they are attached to the root of the hierarchy. MAP is also a function attached to any collection : MAP(C,f), where C is a collection and f a function defined on the elements of C, applies f to each element of C. MAP is a powerful function for processing all elements of a collection. Other functions are specific to a collection subtype (e.g., set). For each subtype of collection x, there exists a conversion function translating a collection y in that subtype of collection : the name of the function is Tox (e.g., ToSet, ToList, ToVector). For example, ToSet(y), where y is a bag, transforms the bag y in a set (thus, it removes

duplicate from the bag). There exists also a Make function, which creates a collection from an enumeration of elements (e.g., MakeSet).

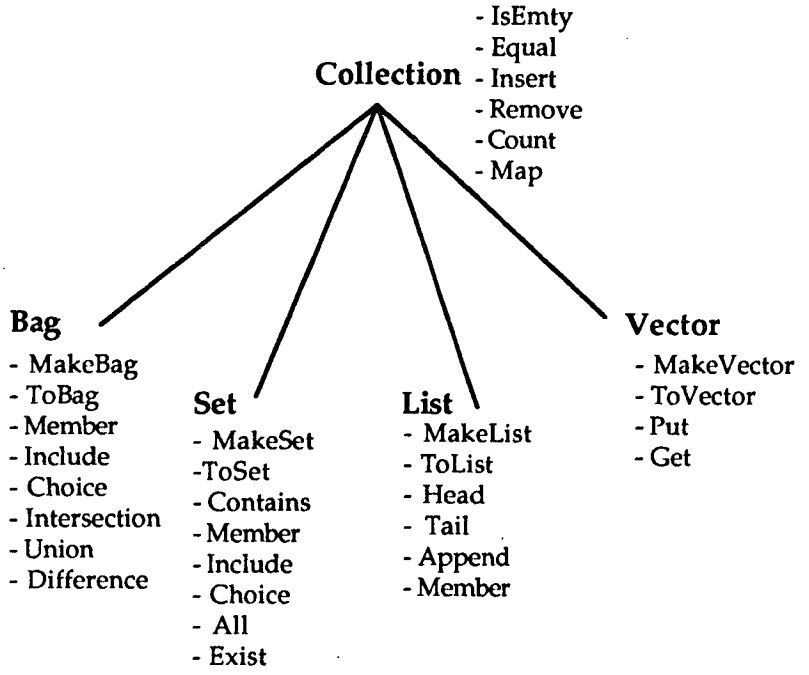

Figure 1 : The Collection generic ADT hierarchy.

A good example of a generic ADT subtype of collection is the set ADT. Denoting "Element" a value of type <Type> and given a function "Function" defined on elements of type <type>, it may be specified as in Figure 2 (including inherited functions).

ADT SET OF <Type>

FUNCTION :

ISEMPTY(Set) --> Boolean

EQUAL(Set, Set) --> Boolean

INSERT(Set, Element) --> Set

REMOVE(Set, Element) --> Set

COUNT(Set) --> Integer

MAP(Set, Function) --> Set

MAKESET(Element [{,Element}...]) --> Set

TOSET(Collection) --> Set

CONTAINS(Set, Element) --> Boolean

MEMBER(Element, Set) --> Boolean

INCLUDE (Set, Set) --> Boolean

```
        CHOICE(Set) --> Element U ø
        ALL(Set, Formula) --> Boolean
        EXIST(Set,Formula) --> Boolean
END SET;
```

Figure 2 : The SET generic ADT.

Note that this ADT specification defines any set of data elements and encapsulates it within the given functions. The first functions, up to MAP, are inherited from the collection type. MAKESET is the creation function while TOSET converts a collection into a set. The other functions respectively check whether a given set contains a given element, whether an element is in a set, whether a set is included in another set. The CHOICE function reduces a set by withdrawing an element at random. The ALL function checks whether all elements of a set satisfy a given formula. The EXIST function returns True if one element in the set satisfy the given formula, and False otherwise. Other functions can be added (e.g., union, intersection and difference of sets). The set generic ADT can be used to define, for example, a set of integers, a set of tuples or a set of complex objects. All these functions can be invoked in ESQL queries as shown below.

3.3 Constructed Type Definition

The specification of a new type in ESQL requires the use of the CREATE TYPE statement. This statement is built from generic types, which are specialized and nested. It introduces a new type by associating a name to a possibly nested type structure. Nesting is possible by value or by reference. Also, a type can be implemented in a relation as a value or an object identifier; in the latter case, objects are stored in an object store associated with the object type. In the case of an ADT that is a specialization of a previously user-defined ADT, the user must provide its supertypes. Additional attributes may be given when the ADT is a specialization of a tuple type. A simplified syntax of the CREATE TYPE command is given in Figure 3. Note that VALUE is the default option for a type specification (type clause) while TUPLE OF is the default option for a structure specification (structure clause).

```
<type definition>   ::=   CREATE TYPE <type name>
                          [SUBTYPE OF <type name> , [{<type name>}...] ]
```

```
                    [ <type clause> ]

<type clause>       ::=     [VALUE] <structure clause>
                        |   OBJECT <structure clause>

<structure clause>  ::=     [TUPLE OF] ( <column name> <type spec>,
                                    [ { <column name> <type spec> } ...] )
                        |   <generic type> OF <type clause>
                                [ (, <type clause> )...]
                        |   < data type >

<generic type>      ::=  SET | BAG | LIST | VECTOR | <identifier>

<data type>         ::=     <character string type>
                        |   <exact numeric type>
                        |   <approximate numeric type>
```

Figure 3 : Syntax of the type definition command.

The following example defines two tuple types ITEM and CUSTOMER, a set type ORDERED. We assume that the ECU type is simply a sub-type of FLOAT. Note that CUSTOMER is an object type, which means that customer instances may be referred to by object identifiers in relations. Objects may be referentially shared between multiple relations.

```
CREATE TYPE ITEM
    (INUM INT, INAME  ARRAY OF CHAR, PRICE ECU) ;

CREATE TYPE CUSTOMER OBJECT
    (CNUM  INT, CNAME ARRAY OF CHAR,
    CPHONES  SET OF INT) ;

CREATE TYPE ORDERED
    SET OF (OITEM  ITEM, QTY  INT) ;
```

The next example defines a polygon type as a tuple object composed of a numerical identifier (PID) and a vertex attribute itself composed of a list of points, and then a triangle type as a subtype of polygon with a height. The

addition of a new attribute to a subtype of a tuple type is illustrated with the height example. Note that an integrity constraint should specify that a triangle has only three vertices.

```
CREATE TYPE POLYGON OBJECT
    TUPLE OF (PID INT, VERTICES LIST OF POINT) ;

CREATE TYPE POINT
    (X FLOAT, Y FLOAT) ;

CREATE TYPE TRIANGLE
    ISA POLYGON
    (HEIGHT FLOAT) ;
```

The next example is more involved. It defines a text as a tuple composed of a title, a set of author names defined as arrays of characters (an alternative is to use the CHARACTER VARYING type of SQL) and a content defined as a list of sections. A section is defined as a number, a title and an array of paragraphs, each represented as a list of phrases. A phrase is a list of references to words in a dictionary. Words are simply arrays of characters.

```
CREATE TYPE TEXT
    TUPLE OF (    TITLE PHRASE,
                  AUTHORS LIST OF ARRAY OF CHAR,
                  TCONTENT LIST OF SECTION ) ;

CREATE TYPE SECTION
    TUPLE OF (    NUMBER INT,
                  TITLE PHRASE,
                  SCONTENT ARRAY OF LIST OF PHRASE) ;

CREATE TYPE PHRASE LIST OF WORD ;

CREATE TYPE WORD OBJECT ARRAY OF CHAR ;
```

3.4 Implementation of ADT Functions

A user constructed type is a specialization of a generic type. As such, it inherits all the functions of the generic types. For example, the type TEXT being a tuple, it inherits the projection function. If t is a text, then TITLE(t) is a phrase and AUTHORS(t) is a list. By applying the FIRST function to the list AUTHORS(t), we obtain FIRST(AUTHORS(t)), which delivers an array of characters or the empty value. This capability of specializing generic types provides a powerful functional language to manipulate nested objects. However, it is not sufficient to express general functions, which should be provided by a general-purpose programming facility.

The support of multiple languages to specify ADTs is provided by the concept of ADT view. An ADT view is a data structure accessible from a programming language, which represents the constructed database ADT as seen in the considered programming language. Of course, the functions inherited by the ADT structure (e.g., TITLE, AUTHORS and FIRST) are callable from the considered programming language.

The general command to define a new function is as follows :

CREATE FUNCTION <function name>
(<operand data type list>)
RETURNS <result data type>
LANGUAGE <language>

While language can be C++, LISP, PROLOG or C, a function specification is a function name followed by the parameter types, the key word RETURNS and the result type. For example, the addition of the surface function (returning a float) to the TRIANGLE data type requires the command :

CREATE FUNCTION SURFACE
(TRIANGLE)
RETURNS FLOAT
LANGUAGE C++ ;

3.5 Type Alterations

More generally, the ALTER TYPE statement enables one to modify an existing named type, for instance, adding a new attribute to a tuple type. In the case of an ADT, the user can change existing operations or modify the subtype relationship. It can also drop a function using the DROP FUNCTION statement or delete an existing type using the DROP TYPE statement.

4. TABLE CREATION AND MANIPULATION

4.1 Table Creation

As previously mentioned, the basis of the ESQL data model is relational, generalized for supporting collections of objects and values. Collections of objects or values are recorded in set of tuples (i.e., relations), which are the first level constructs (in other words, relations are first class citizens).

The CREATE TABLE statement of SQL is generalized to allow a table to be defined as a collection of tuples of attribute names of possibly complex types. To support referential object sharing, a value type in a tuple definition can be replaced by an object type. Object type is defined at type creation time using the OBJECT keyword. The general syntax of the CREATE TABLE statement is as in SQL2. Note that table inheritance is supported in SQL2. We give here a simplified syntax for it.

<table definition> ::=
    CREATE TABLE <table name>
        [**SUBTYPE OF** <table name> [{, <table name>}...]]
[(<table element> [{ , <table element> } ...])]

<table element> ::=
        <column definition>
    | <table constraint definition>

<column definition> ::=
        [<column name>] <data type>
        [<column constraint definition>]

The CREATE TABLE statement of SQL is basically unchanged. The only modification is that a data type can now be an ADT (user-defined or built-in) and attribute names in column definitions are optional. Note that a table can inherits attributes from existing tables. Also, attribute names can be omitted in the case of a tuple data type. Attribute names are then inferred from the tuple attributes.

Examples of table creation are given below :

```
CREATE TABLE ITEMS
    ( ITEM )
    KEY IS INUM ;

CREATE TABLE CUSTOMERS
    (CUST  CUSTOMER)
    KEY IS CUST ;

CREATE TABLE ORDERS
    (ONUM  INT, CUST CUSTOMER, ORDER  ORDERED)
    KEY IS ONUM ;

CREATE TABLE NEWORDERS
    SUBTYPE OF ORDERS
    (PRIORITY INT) ;
```

Note that the CUSTOMER object tuples can be referentially shared between the CUSTOMERS relation and the ORDERS relation. The ORDERED set is nested by value within the ORDER relation. NEWORDERS inherits the ORDERS attributes and is extended with a priority attribute.

```
CREATE TABLE TEXTS
    ( CONT TEXT)
    KEY IS TITLE(CONT) ;
```

creates a relation of texts stored by value. Note that the key is defined using a function applied to the text type. In the next section, further use of functions will be exemplified.

```
CREATE TABLE TRIANGLES
```

```
( RNUM INT, COLOR STRING, SHAPE TRIANGLE )
KEY IS RNUM ;
```

creates a relation of colored triangles whose shape is stored by reference.

4.2 Querying tables with Single-valued Attributes

The ESQL language is an upward compatible version of SQL. The user sees relations and can manipulate them using SQL. However, since relations may be defined over complex domains, it is also possible to manipulate data inside a complex structure using the ADT functions. Thus, the first significant extension is the possibility of ADT operations. Complex structure instances are encapsulated by their functions as specified in the type definition. In general, any function attached to a given type may be applied to an element of that type. If A is of type T1 and F is defined over type T1 as producing type T2 (i.e., F(T1) --> T2), F(A) may be used in any expression of type T2. If G is defined over type T2 as producing type T3, G(F(A)) may be used in expressions of type T3. Thus, SQL2 expressions are generalized to include the possibility of applying ADT functions at various levels. Expressions and sub-expressions must be correctly typed. Of course, functions that update objects or values cannot be used in queries (SELECT statement) but only in updates (UPDATE statement).

ESQL data manipulation using user-defined or generic functions is now illustrated. For example, let FRANCS be an ADT user function converting ECU into French francs expressed in the float data type (FRANCS(ECU) gives FLOAT). The following query on the ITEMS relation lists all items for which the price is less than 100 francs.

```
SELECT    *
FROM      ITEMS
WHERE     FRANCS(PRICE) < 100 ;
```

To deal with objects, a VALUE function is necessary to read through an object identifier. However, this function can be implicitly applied by the system to get the correct typing. Consequently, the following query returns the name of the customer whose order number is 10.

```
SELECT    CNAME(CUST)
FROM      ORDERS
WHERE     ONUM = 10 ;
```

This is a short notation for :

```
SELECT    CNAME(VALUE(CUST))
FROM      ORDERS
WHERE     ONUM = 10 ;
```

Conversely, the manipulation and updating of object identifiers may require accessing the reference of an object X using the OID function. OID(X) gives the object identifier of X. The OID function makes precise the distinction between object equality and object identity. Given objects O1 and O2, O1 = O2 will return true if the value of the objects are equal whereas OID(O1) = OID(O2) will return true if they both refer to the same object identifier.

4.3 Querying Collections of Objects

Built-in functions can be used to manipulate collections of objects, which are specific generic ADTs instances. Functions can be composed in expressions. To simplify the handling of collections in ESQL, a function F defined on a type T and not defined on a collection of T (denoted here Collection(T)) can be directly applied to a collection of T in ESQL queries. The semantics of such an application is to apply F to all elements of the collection and return the collection of results. Thus, F(Collection(T)) is simply Collection(F(T)). This corresponds to an automatic generation of the MAP function; formally, F(Collection(t)) is rewritten into MAP(F, Collection(T)). For example, the following query lists the names of the customers which order a disk. Note that an attribute function is applied directly to a set of tuples having that attribute ; such an application returns the set of attribute values of the tuples in the set.

```
SELECT    CNAME(CUST)
FROM      ORDERS
WHERE     CONTAINS(INAME(OITEM(ORDER)), "Disk") ;
```

Note that CONTAINS is the Boolean function defined on sets to determine whether an element (here "Disk") belongs to the set. OITEM and INAME are projection functions directly applied to set of tuples.

Assuming that the list data type can also be manipulated using a MEMBER function (i.e., MEMBER(Element, LIST) --> Boolean), the following query retrieves the first author of all texts containing the word "Database" in the title.

```
SELECT   FIRST(AUTHORS(CONT))
FROM     TEXTS
WHERE    MEMBER(,"Database",TITLE(CONT)) ;
```

Note that the query assumes an implicit translation of OIDs to values, as TITLE(CONT) is in fact a list of word identifiers. A complete qualification without implicit notations would be :

MEMBER("Database", VALUE(TITLE(CONT))).

The value function is useless since an automatic translation from OIDs to values is assumed when necessary.

Collection handling requires the ability to transform single-valued attributes to collection, according to a grouping criteria. This is the classical "nest" operation, generally implemented by a group by in SQL. In ESQL, we maintain the GROUP BY clause. However, the result of a group by does not require applying an aggregate function (e.g., SUM) : it can simply be a collection. By default, we assume that the bag collection is specified. Thus, the following query gives a bag of large triangles (surface greater than 10) for each color :

```
SELECT COLOR, SHAPE
FROM TRIANGLES
WHERE SURFACE(SHAPE) > 10
GROUP BY COLOR
```

To get another type of collection (e.g., a set), a conversion function (e.g., TOSET) must be applied. Thus, the following query returns set of large triangles per color :

```
SELECT COLOR, TOSET(SHAPE)
FROM TRIANGLES
WHERE SURFACE(SHAPE) > 10
GROUP BY COLOR
```

It is also possible to unnest a relation with a multi-valued attribute, according to an ungrouping attribute. This is performed by an UNGROUP clause, similar to the GROUP BY clause of SQL. For example, the following query gives each product ordered for each customer. :

```
SELECT CUST, ORDER
FROM ORDERS
UNGROUP BY CUST
```

4.4 Updating Tables

The UPDATE command of SQL can now be used to modify objects. For example, one can add "John Doe" as author of all texts containing the database keyword in the title using the following command. Note that the "=" of SQL is replaced by ":" as the command does not assign a new value to a complex attribute (here CONT), but rather applies a function to it.

```
UPDATE TEXTS
    SET CONT :  INLIST(AUTHORS(CONT), ["John Doe"])
    WHERE       CONTAINS(TITLE(CONT), W)
    AND         VALUE(W) = "Database" ;
```

In summary, the nesting of ADT functions in ESQL expressions provide a concise and powerful tool for manipulating any nesting of generic or user defined ADTs. It is a regular extension of SQL, which reduces to the normalized version for flat tables.

5. DERIVED TYPES AND RELATIONS

Much work has been devoted to extending relational databases to deductive databases. The so-called DATALOG language allows the user to derive virtual relations from the database. DATALOG has the power of

relational algebra plus recursion. It may be extended to support negation and functions. Unfortunately, the syntax of DATALOG does not fit well with that of SQL. However, given their importance, recursive queries have been introduced in SQL3 but are not easy to express.

ESQL supports deductive capabilities through derived relations defined as generalized views. We simply extend the view concept of SQL2 with recursion, by allowing a view to be defined from itself. Deriving a relation from a relation using a query expression referencing this relation is a powerful functionality. As an example, consider the type PARTS(PART ITEM, COMPONENT ITEM). The following recursive view definition specifies the transitive closure of the PARTS relation :

```
CREATE VIEW SPARTS AS
        SELECT *
        FROM PARTS
UNION
        SELECT P1.PART, P2.COMPONENT
        FROM SPARTS P1, PARTS P2
        WHERE P1.COMPONENT = P2.PART ;
```

The form of the relation derivation statement which gives the power of standard DATALOG is :

```
CREATE VIEW <table name> [<column name>]...
        AS <query>
        [ UNION <query> ] ... ;
```

where <query> is an SQL query. The query can refer to the derived table, thus allowing the database administrator to define recursive relations. Furthermore, ADT functions can be used in queries, thereby giving ESQL the power of DATALOG with functions.

Let us assume a ROADS relation, representing the roads between large cities in EUROPE, created as follows :

```
CREATE TABLE ROADS ( #R INT, SCITY STRING,
                     TCITY STRING, LENGTH FLOAT ) ;
```

An example of a derived table with functions is a PATHS relation, which gives all paths from one city to another with their lengths. It is defined as follows :

```
CREATE VIEW PATHS (SCITY, TCITY, LENGTH) AS
        SELECT SCITY, TCITY, LENGTH
        FROM ROADS ,
UNION
        SELECT P.SCITY, R.TCITY, P.LENGTH + R.LENGTH
        FROM PATHS P, ROAD R
        WHERE P.TCITY = R.SCITY ;
```

To support negation as in certain versions of DATALOG, we use the EXCEPT command of SQL2. For example, if one wants to avoid all paths going through special roads recorded in a FORBID relation, we can define the following view :

```
CREATE VIEW PATHS (SCITY, TCITY, LENGTH) AS
(       SELECT SCITY, TCITY, LENGTH
        FROM ROADS
EXCEPT
        SELECT SCITY, TCITY, LENGTH
        FROM FORBID  )
UNION
        SELECT P.SCITY, R.TCITY, P.LENGTH + R.LENGTH
        FROM PATHS P, ROAD R
        WHERE P.TCITY = R.SCITY ;
```

ESQL supports stored queries within relations [Stonebraker86] by allowing a type to be defined as a query. Deriving a type enables the database administrator to create a new type from a more or less complex query. The syntax of the type derivation statement is :

```
CREATE TYPE <type name> AS <Query> ;
```

It can be used for example to implement truly nested relations as in the following example :

```
CREATE TYPE NORDER AS
    SELECT *
    FROM ORDERS
```

and then :

```
CREATE TABLE NESTEDORDERS
(NUM : INT, TAB : NORDER) ;
```

6. A FUNCTIONAL SEMANTICS FOR ESQL

The functional semantics of ESQL is operational in nature. The principle is to translate an ESQL query or update into a functional expression using the primitive ADTs. Thus, a program that computes the semantics of the query can be obtained. For this purpose, we introduce the relation data type and additional programming constructs (while, if). Next, we give an algorithm to translate a query or an update into an equivalent operational program.

6.1 The Relation Data Type

A relation can be defined as a generic ADT built from the set and tuple generic ADTs, as follows :

ADT SET OF TUPLE OF (ATTRIBUTE : <Type>, ...).

It can then be manipulated using functions on sets (e.g., ISEMPTY, INCLUDE, EXTRACT, etc.) and functions on tuples (PROJECT, CHECK, etc.). To simplify the manipulation of relations, we define new functions corresponding to the classical relational operators. These functions can be specified in terms of the SET and TUPLE generic ADTs.

The ability of checking formulas and performing operations on sets imposes the introduction of two programming construct whose semantics are well known. The "**while <condition> do <function>**" construct performs the function while the condition is true and does nothing otherwise. The "**if <condition> then <function1> [else <function2>]**" construct performs function1 if the condition is true, and function2 (or nothing) otherwise.

Using these two constructs and the function mapping on sets, it is possible to define in a simple fashion the classical relational algebra operators on relations (i.e., set of tuples). We obtain the following ADT .

```
ADT RELATION OF (ATTRIBUTE : <Type>, ... )
   FUNCTION :
      PROJECT(Relation, A1, A2,...) --> Relation
      RESTRICT(Relation, <Formula>) --> Relation
      PRODUCT(Relation, Relation) --> Relation
      UNION(Relation, Relation) --> Relation
      DIFFERENCE(Relation,Relation) --> Relation
END RELATION;
```

For example, the project function of a relation R into a relation S can be defined as follows :

```
PROJECT(R, A1, A2,...) =
        while NOT ISEMPTY(R) do
            INSET(S,(PROJECT(EXTRACT(R),A1,A2,...))
```

Similarly, the restrict function can be defined as follows :

```
RESTRICT(R , F) =
        while NOT ISEMPTY(R) do
            if CHECK(NEXT(R),F)) then INSET(S,EXTRACT(R))
```

Another alternative for set manipulation is to introduce an implicit mapping of functions on sets to functions on elements, as previously mentioned. This can be done by interpreting a function applied to a collection (i.e., F(Collection(e1,e2, ...))) as a collection of function applications (i.e., Collection(F(e1), F(e2), ...)). For example, a function applied to a set of elements is interpreted as a set of elements resulting from the application of the function to each element of the set. If the function is undefined on an element, an undefined element is added to the set. Using the facility which maps a function on a collection to a similar collection of functions on the attributes, the restrict and projection functions can be defined without while constructs.

6.2. Translating ESQL Requests into Functional Programs

We illustrate the technique on an ESQL query. Let us assume the query:

```
SELECT   CNAME(VALUE(CUST))
FROM     ORDERS
WHERE    CONTAINS(INAME(OITEM(ORDER)), "Disk")
```

First, the where clause is translated into a functional expression :

RESTRICT (ORDERS, CONTAINS(INAME(OITEM(ORDER)), "Disk")

Next, the SELECT clause is interpreted as a PROJECT function, which gives :

PROJECT (CNAME(VALUE(CUST)),RESTRICT (ORDERS,
CONTAINS(INAME(OITEM(ORDER)), "Disk")).

To remove operations on relations, we can replace the RESTRICT and PROJECT operation by their definitions seen in the previous section. We then obtain a program built from primitive data types.

7. CONCLUSION

In this paper, we have extended standard SQL for better supporting new application domains. The main extensions are twofold : the support of abstract data types and the support of recursive rules as extended views. This demonstrates the integration in a common abstract data type framework of the relational, object oriented and deductive paradigms. The integrated approach is based on the introduction of nested generic abstract data types, the relation abstract data type being the first visible data type. Compared to previous proposals [Beech88, Carey88, Gardarin89b, Scheck88, Valduriez88a], ESQL goes further in terms of nested data types, object oriented and deductive capabilities. Furthermore, the integration of object oriented features is only based on a generic ADT framework, which is easy to implement.

Furthermore, we introduced a functional operational semantics for ESQL. This semantics gives a first translation of a query into a functional program. This program could be used as a basis for query optimization using

[ISO86] ISO ANSI, "Database Language SQL", ISO/DIS 9075, International Standard, 1986.

[Khoshafian86] S. Khoshafian, G. Copeland, "Object Identity", Int. Conf. on OOPSLA, Portland, Oregon, September 1986.

[Khoshafian87] S. Khoshafian, P. Valduriez, "Persistence, Sharing and Object Orientation: a database perspective", Int. Workshop on Database Programming Languages, Roscoff, France, September 1987.

[Melton89] J. Melton, " Database Language SQL2 and SQL3", ISO-ANSI working draft, ISO DBL CAN-2b, May, 1989.

[Ozsoyoglu87] Z.M. Ozsoyoglu, L-Y. Yuan, "A New Normal Form for Nested Relations", ACM TODS, Vol. 12, No. 1, March 1987.

[Schek88] H. Schek, "Nested Relations, a Step Forward or Backward", IEEE Data Engineering, Vol. 11, No. 3, September 1988.

[Stonebraker83] M. Stonebraker, B. Rubenstein, A. Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Databases", ACM SIGMOD Int. Conf., San Jose (California), May 1983.

[Stonebraker86] M. Stonebraker, L.A. Rowe, "The Design of POSTGRES", ACM SIGMOD Int. Conf., Washington, D.C., May 1986.

[Valduriez89a] P. Valduriez, S. Danforth, "Functional SQL (FSQL), an SQL Upward Compatible Database Programming Language", MCC Technical Report ACA-ST-045-89, February 1989, to appear in Information Sciences.

[Valduriez89b] P. Valduriez, S. Danforth, B. Hart, T. Briggs, M. Cochinwala, "Compiling FAD, a Database Programming Language", Int. Workshop on Database Programming Languages, Portland, Oregon, June 1989.

[Valduriez89c] P. Valduriez, S. Danforth, "Query Optimization in Database Programming Languages", Int. Conf. on Deductive and Object Oriented Databases", Kyoto, Dec. 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

transformation rules. Other techniques are possible for query optimization [Valduriez89c].

ESQL is the interface to the database system of the European Declarative System (EDS) developed in an ESPRIT.II project (1989-1992). ESQL is to be compiled and optimized for parallel execution on the EDS highly parallel data server. ESQL applications primarily include both on line transaction processing queries and complex decision support queries for large expert systems.

Further enhancements to ESQL are planned in a near future. This includes the addition of conditional and loop statements to program multi-statement procedures and perform explicit loops on set elements. Another planned extension is triggers to manage active databases. We also plan to add a good support for integrity constraints, both on tables and ADTs.

ACKNOWLEDGMENTS

We would like to thank all our colleagues in the EDS ESPRIT project for their careful reading of ESQL drafts and thoughtful comments and suggestions.

REFERENCES

[Bancilhon88] F. Bancilhon, "Object-Oriented Database Systems", Int. Symp. on PODS, Austin, Texas, March 1988.

[Beech88] D. Beech, "A Foundation for Evolution from Relational to Object Databases", Int. Conf. on EDBT, Venice, Italy, March 1988.

[Carey88] M.J. Carey, D.J. DeWitt, S.L. Vandenberg, "A Data Model and Query Language for EXODUS", SIGMOD Int. Conf., Chicago, Illinois, June 1988.

[Danforth88] S. Danforth, P. Valduriez, "The Data Model of FAD, a Database Programming Language, Rev.1", MCC Technical Report ACA-ST-059-88, June 1988, to appear in Information Sciences.

[Gardarin89a] G. Gardarin, P. Valduriez, "Relational Databases and Knowledge Bases", Book, Addison-Wesley, 1989.

[Gardarin89b] G. Gardarin et al., "Managing Complex Objects in an Extended Relational DBMS", Int. Conf. on VLDB, Amsterdam, August 1989.

[Guttag77] J. Guttag, "Abstract Data Types and the Development of Data Structures", Comm. of ACM, Vol. 20, No. 6, June 1977.